FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Automatic Performance Analysis for Shared Virtual Memory Systems

Michael Gerndt, Andreas Krumme

KFA-ZAM-IB-9631

Oktober 1996
(Stand 21.10.96)

Erscheint in: Proceedings ”CPC’96 - Compilers for Parallel Computers 1996”, Aachen, 11.-13.12.1996

Automatic Performance Analysis for Shared Virtual Memory Systems

M. Gerndt, A. Krumme
Research Centre Jülich (KFA)
Central Institute for Applied Mathematics
52425 Jülich, Germany
{m.gerndt, a.krumme}@kfa-juelich.de

Abstract

Programming distributed memory multiprocessors requires program parallelization as well as program optimization with respect to data locality. SVM-Fortran is a programming language for shared virtual memory architectures with special language features for specifying the distribution of parallel tasks onto the processors. It is realized on top of a shared virtual memory implementation on Intel Paragon. A programming environment provides performance analysis tools helping the user in the optimization of data locality. This paper outlines the environment, describes the basic concepts of the performance analysis support, and presents a design for the automation of performance analysis.

1 Introduction

Parallelization of sequential programs for massively parallel computers with distributed memory is simplified by languages providing a global name space. SVM-Fortran (SVMF) is a shared memory parallel programming language based on Fortran 77. It provides additional language features for scheduling loop iterations and parallel sections onto the processors. The target architectures of SVM-Fortran are distributed memory systems with a global address space and with dynamic replication and migration of data among the processors via local caches or Shared Virtual Memory (SVM) [1]. This data migration allows to optimize data locality by optimizing data reuse which results from a well-done work distribution onto the parallel processors.

The primary target system of SVM-Fortran is the Intel Paragon. The Advance Shared Virtual Memory system (ASVM) is a replacement of the XMM system in the MACH 3.0 micro-kernel and implements the global address space. The SVM-Fortran compiler transforms SVM-Fortran into Fortran 77 code which accesses shared memory objects via the UNIX System V shared memory interface. This code is then compiled by the native Fortran compiler.

Performance analysis tools are required for understanding the run-time behavior of an application. These tools have to be source-code-based and have to provide automatic user guidance to be applied by the users. In addition, detailed information must be

obtainable without forcing the user to suffer from enormous trace files. The lack of solutions for these requirements are at least one important reason for the reluctance of users to apply current tools. The SVM-Fortran programming environment provides performance analysis tools taking into account these requirements via selective tracing in combination with an incremental performance analysis approach. Based on the experiences gained with the Optimizer and Locality Analyzer (OPAL) we designed an automatic analysis module for this tool.

This article outlines SVM-Fortran and its programming environment. We focus on the performance analysis support and present related work in that area in Section 2. Section 3 gives a short overview of the language extension for global work distribution in SVM-Fortran. In Section 4 we describe the incremental analysis concept and its implementation. The design for the automation of the performance analysis procedure is presented in Section 5. The last section summarizes the status of the described work.

2 Related Work

Performance analysis tools for message passing programs, such as Paragraph [2], Pablo [3], and VAMPIR [4], support the user in analyzing enormous amounts of trace data for low-level send/receive operations. The tools allow to filter and visualize trace data after the program run but do not include any knowledge about potential bottlenecks in such programs.

Data parallel languages, e.g. HPF, allow the user to focus on the data distribution. For HPF, profiling tools are available which give coarse information about program performance. Detailed information is only available for the generated message passing code and thus, the relation to the original source code is lost.

The performance analysis tool APPRENTICE [5] for the Craft programming model of the Cray T3D supports the investigation of performance data based on the high-level source code. It provides information on the number of private, local shared, and remote shared load and store operations on the level of basic blocks of the shared memory parallel program. Since the information is only summary information for a whole program run and for all processors it is not sufficient to perform a detailed analysis. In addition, the overhead for performance monitoring is extremely high and thus, the gathered information is only of limited relevance.

The only tool known to the authors providing automatic performance analysis is the Paradyn environment developed at the University of Wisconsin [6]. It applies on-the-fly run-time instrumentation and performance analysis. This tool tries to proof hypotheses on performance bottlenecks by applying appropriate rules, but supports only a very limited number of rules and suffers from performing the analysis while the program is still executing. The investigation has to be done for significant program phases which are unknown to the analysis tool.

3 SVM-Fortran Language Summary

SVM-Fortran [7, 8] is a shared memory parallel Fortran77 extension targeted mainly towards data parallel applications on shared virtual memory systems and distributed

shared memory systems which provide hardware support for a global address space on top of physically distributed memory. It is based on HPF [9], Fortran-S [10], and KSR Fortran [11]. SVM-Fortran supports coarse-grained functional parallelism where the parallel task itself can be data parallel.

The execution model of SVM-Fortran is an extension of the Single-Program-Multiple-Data (SPMD) model. SVM-Fortran provides the standard features of shared memory parallel Fortran languages as well as specific features to determine the distribution of tasks onto processors. Similar to Fortran-S and KSR Fortran, loop annotations can be used to determine a static or dynamic work distribution scheme. Examples are *direct scheduling*, such as BLOCK and CYCLIC, as well as *dynamic scheduling*, e.g. self-service scheduling. *Aligned scheduling* can be used to schedule iterations following the principle of data locality, i.e. execute an iteration on that processor where the data resides.

Data locality is not a problem to be solved on the level of individual do-loops but is a global problem. Therefore, SVM-Fortran borrowed the concepts of processor arrangements and templates from HPF as tools to specify scheduling decisions globally via template distributions. With *predefined scheduling*, loop iterations are assigned to processors according to the distribution of the appropriate template element.

Templates can be handled very flexible. They can be created, distributed and redistributed dynamically at any point in the program, and passed via the subroutine interface. SVM-Fortran supports *standard distributions* (like BLOCK, CYCLIC, and GENERAL_BLOCK), *indirect distribution* and *linked distribution*. The programmer can specify for each template element the target processor by an arbitrary integer expression within an indirect distribution. Linked distribution is a form of alignment where a distribution is described via the distribution of another template.

```
      PARAMETER (n=64000)
      INTEGER work(3), map(n)
CSVM$ PROCESSORS:: p(4)
CSVM$ TEMPLATE:: t1(n), t2(n)

C     -- general block distribution --
      work(1) = 18000
      work(2) = 16000
      work(3) = 22000
CSVM$ REDISTRIBUTE(GENERAL_BLOCK(work))
CSVM$+     ONTO p::t1

C     -- indirect distribution --
      READ(*,*) map
CSVM$ REDISTRIBUTE (i) ONTO p(map(i)):: t2

CSVM$ PDO(STRATEGY(ON_HOME(t2(i))))
      DO i=1,n
         a(ind(i)) = ...
      ENDDO
```

Figure 1: Examples of template distributions and their usage in loop scheduling

Figure 1 gives some examples of distributions. In the first example, the user specifies in a *general_block* distribution how the work (i.e. the loop iterations) should be distributed, i.e. 18000, 16000, 22000, and 8000 template elements are distributed to the 4 processors. In the second example, the user specifies an indirect distribution, the indirection is given with a mapping array `map`. The template can then be used for loop scheduling as can be seen in the parallel loop.

4 Performance Analysis for SVM-Fortran

SVM-Fortran facilitates program development for SVM systems. It provides easy-to-use language features to implement a global parallelization strategy to support the optimization of programs with respect to data locality. The optimization, which is an important task in the incremental parallelization of programs, is supported by a performance analysis environment. This environment consists of the performance monitoring support integrated into the ASVM, the SVM-Fortran Application Monitor (SAM) [12], and the performance analysis tools OPAL and PARvis.

4.1 Performance Bottlenecks

Performance analysis aims at detecting performance bottlenecks in a program. Either the user or the performance analysis tools have to understand the potential bottlenecks typical for SVM-Fortran programs and the information required to identify these bottlenecks in a program run.

Potential bottlenecks in SVM-Fortran programs mainly consist of:

- missing data locality
- load imbalances
- synchronization
- SVM-Fortran administration

Although each overhead type can be crucial, the data locality bottleneck is used in the remainder of the article to simplify the description of the analysis concepts.

Data locality bottlenecks can be identified in SVM systems in form of the page transfers among the processors. The reasons for pagefaults are manifold: A pagefault occurs when the page is first accessed in a processor, pagefaults also occur due to capacity problems and coherence operations. While the first access to pages is not critical for program performance, capacity problems, and coherence problems can lead to enormous paging overhead.

Capacity problems in the processors mainly have three reasons. First, regions of the code may not be parallelized and thus a single processor accesses the whole data structure. Second, the processor requires more data for the computation of its part of the application domain than fits in its local memory. Third, the work distribution of some parallel loops is not based on the global parallelization strategy and thus, unnecessary access to other parts of the data structures occur.

pagefaults due to coherence operations result from two main reasons: true sharing and false sharing. While true sharing results from accesses to the same data in different processors, false sharing occurs when different data are accessed which are laid out on the same page. Especially false sharing is frequently the reason for page thrashing, i.e. multiple exchanges of the same page among the same pair of processors.

4.2 Incremental Analysis

Besides the detection of the existence of a bottleneck, the user has to determine the exact area of code and the reason for the bottleneck. This requires very detailed information, e.g. the existence of a data locality bottleneck can be proven based on the amount of pagefault time in the whole program, but the identification of the location requires pagefault information for individual program regions and program variables.

Since the approach of generating detailed information for the whole program run and selecting useful information afterwards in the analysis tool leads to enormous trace files, the SVM-Fortran programming environment is based on selective tracing and on the incremental performance analysis concept outlined in Figure 2.

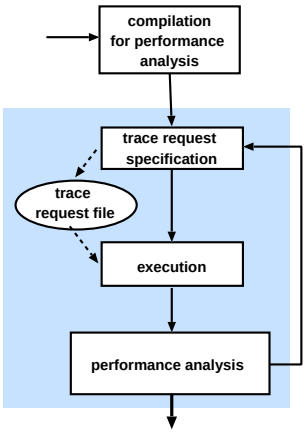

Figure 2: Incremental performance analysis

Performance monitoring is guided by a trace request file. The requests are handled either directly by SAM or SAM instruments the ASVM to trace information only available for the kernel. Trace data are stored in a trace buffer which is flushed at barriers to reduce the effect of program perturbation.

Based on this selective tracing, the user can first request coarse-grained information and then request more specific information in interesting program regions based on the previously gathered information. This incremental approach reduces the amount of trace data by taking into account already known information.

4.3 Trace Request File

The user specifies in the trace request file which information should be gathered at run-time. The requests are formulated by using routine and variable names from the source code. Examples for such requests are:

request (∗) *local *.reg_no(*): RPFsum, WPFsum*
request (∗) *local foo.reg_(5): RPFsum(A,B)*
request (∗) *local foo.reg_no(5): RPF(A)*

The first request specifies that in all processors and all regions, i.e. parallel loops, parallel sections, and subroutines, the sums of the read and write pagefaults have to be

traced. Thus, for each instance of a region the information including an identification of the region is written to a trace file. This demonstrates two features of the SVM-Fortran trace format: source code support and the hierarchy of trace information. The requested sums allow to determine regions with high pagefault rates and, more precisely, the incarnations with high pagefault rates.

The second request determines for a specific region in subroutine `foo` the individual pagefault sums for specific arrays. This typically leads to the array responsible for a large number of page faults.

The last request generates individual events for each pagefault. This precise information is used in the analysis of the program behavior to determine exactly the array element accessed when the pagefault occurred.

4.4 Program instrumentation

The concept of incremental analysis requires a flexible monitoring support. The monitoring system has to provide appropriate information at different levels of granularity. Only if coarse- and fine-grained information is supported, the user can make appropriate decisions.

The performance analysis monitor SAM provides such levels in three different areas:

1. program regions

   The instrumentation can be requested for individual program regions, such as subroutines, parallel loops and parallel sections. On the coarsest level, information is generated only for the whole program, on the finest level for each region.

2. execution time

   SAM provides different resolutions with respect to the execution time. Information can be gathered as statistics written at the end of the program run summarizing the information for all instances of a region in a process. SAM also provides sampling, i.e. the information is generated for every n-th execution of a region thus providing an overview of the behavior of multiple instances. The most precise information results from generating data for each instance.

3. information detail

   The trace format supports a hierarchy of trace events [13]. This hierarchy is outlined for the area of data locality:

   (a) read-page-fault-sum, write-page-fault-sum, write-upgrade-sum, page-in-pager-sum, page-out-pager-sum, page-out-node-sum, page-discard-sum:

   The sums for read and write pagefaults as well as for paging activity due to capacity problems are the coarsest information.

   (b) page-travel:

   This event determines the number of page exchanges among pairs of processors.

   (c) read-page-distribution, write-page-distribution:

The page distribution when a parallel loops starts and terminates can give a good overview of the individual page faults without generating individual records for each pagefault.

(d) read-page-fault-serviced, write-page-fault-serviced, page-in, page-out, page-discard, reduce-access-permission, invalidate-copy:

These are the most detailed trace records generated for individual pagefaults.

In addition to tracing run-time information, SAM provides features to gather information on the overhead induced by the tracing itself. This information is necessary to decide, whether the measured information reflects the actual program behavior.

4.5 The Performance Analysis Tool OPAL

OPAL (Optimizer and Locality Analyzer) [14] supports a menu-driven specification of trace requests that frees the user from the subtle specification syntax. In addition, OPAL analyzes trace data and extracts the most useful statistical data, but does not store all the trace information in memory. These statistical data can be visualized as annotations to the source code in a separate performance column in the main window. This way of presenting information does provide a good overview of the performance results.

If the user needs more information on individual trace regions, he can mark a trace region and request a list of all trace events of a specific type. For example, OPAL presents all shared variables accessed in the selected trace region, and the user can select specific variables for which he would like to see individual pagefaults. The tool then extracts from the trace file of an individual processor only the pagefault information for these variables and the selected program region. Due to the source code information in the trace files, all the information can be requested and presented in form of trace regions and program variables, e.g. the faulting address is translated into the array name and the indices of the array element.

Thus, the tool allows to instrument selected regions of the source code to obtain certain information. Starting from statistical data for these program regions, incarnation-specific information can be analyzed for regions of the source code.

The major drawback of such an environment is that still the user has to navigate through the large number of analysis choices and has to detect bottlenecks by himself. Future tools have to include user guidance as well as the ability to automatically analyze program performance. For frequently occurring performance bottlenecks, the tool should be able to automatically proof the existence of these bottlenecks.

5 Automation of Performance Analysis

Experiences with the tool OPAL in optimizing the performance of SVM-Fortran applications build the basis for an automatic rule-based optimizing approach. One experience is that the optimizing approach in sequential programs, namely concentrating on the regions with the highest execution time, can be applied also in a slightly different way to parallel programs. In order to judge the successfulness of the parallelization, the

user computes the speedup between the sequential and the parallel program. In the case of insufficient speedup he concentrates on the regions with the highest absolute total overhead, calculated as follows:

$$\text{absolute total overhead} = \text{parallel execution time} - \frac{\text{sequential execution time}}{\text{processor count}}$$

The analysis of different applications has shown that programs fall between the following two extremes

1. few regions with a high overhead time
2. many regions with a comparable overhead time

Clearly the first item is easier to analyze than the second. But this difference does not influence the choice of an automatic analysis method for which two approaches are considered.

The first approach assumes that an execution of the sequential version on one processor and the parallel version on more processors can be carried out. The great advantage of this approach is the detection of all overheads, even if they cannot be traced. The absolute overhead time consists of a portion with already traceable overhead types like pagefaults and synchronization and a portion of unknown or untraceable overhead types like cache misses. With this approach, no time is wasted with the analysis of regions with a large traceable but relatively small total overhead and the user can detect new overhead reasons that should be integrated into the automatic performance analysis.

If the sequential run cannot be performed, the second approach will be chosen. The user concentrates on the region with the highest traceable overhead. Unknown and untraceable overhead cannot be detected and thus no knowledge about how much overhead could be spared in total is achieved.

In the reminder of this article, we assume that the sequential run is possible. After the description of the general optimizing approach, the next section will discuss the implementation concepts of a rule-based automation of the performance analysis.

5.1 Design of an Automatic Analysis Module

This section outlines the components of an *automatic analysis module* (Figure 3) for the SVMF environment. This module analyzes the available performance information, i.e. run-time information and analysis information determined from the source code, and tries to proof hypotheses about performance bottlenecks. If a hypothesis can be proven it is output as a *performance bottleneck*. A hypothesis is output as a *bottleneck hint* if it has a probability below 100% and more refined rules are tested negative or no rule exists which might have a higher probability. The probability is also output as part of the hint.

To proof other hypotheses, more run-time information may be necessary. In this case the analysis module creates new information requests. The information requests identify all useful information and the instrumentation synthesis has to decide which instrumentation to perform during the next program run. This decision takes into account the execution time of the program and the possible perturbation due to the instrumentation. For example, it can predict the amount of perturbation caused by instrumenting

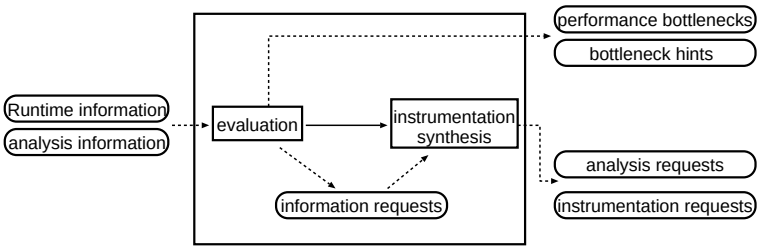

Figure 3: Structure and interfaces of automatic analysis module

the pagefaults in a program region based on the pagefault counts available for that region.

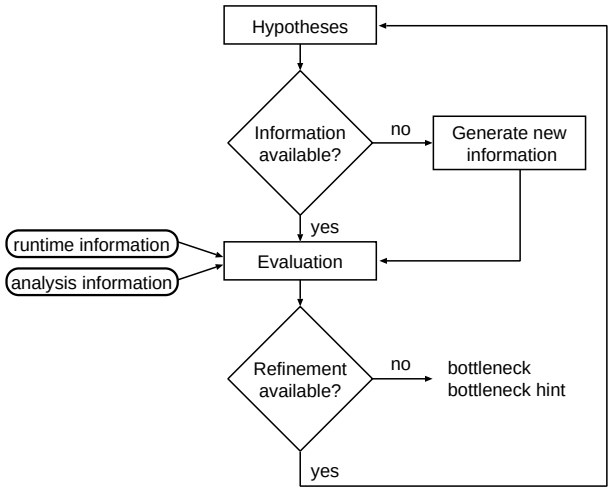

Figure 4: Hypothesis evaluation and refinement

Figure 4 gives a detailed overview of the evaluation phase. The whole process is based on two sets of rules: refinement rules and proof rules. In a first step, the current set of hypotheses is evaluated based on the proof rules and the available performance information. The proof rules contain knowledge about necessary performance information and are used to test a hypothesis. The evaluation may lead to performance bottlenecks and bottleneck hints as described earlier. If refinement rules are available for a proven hypothesis a number of new hypotheses is generated which are more detailed than the current hypothesis. After the refinement step, the applicable proof rules are determined and the required information is requested.

The analysis support module is based on knowledge about the potential performance bottlenecks and the information required to proof the bottlenecks. The knowledge is implemented as a rule base. This implementation leads to a good documentation of the available knowledge and can be easily extended.

Rule	hypothesis		refined hypotheses
1	TRUE	$\Longrightarrow$	bottleneck
2	bottleneck	$\Longrightarrow$	bottleneck caused by unmeasured overhead bottleneck caused by measured overhead
3	bottleneck caused by measured overhead	$\Longrightarrow$	locality bottleneck load balancing bottleneck synchronization bottleneck SVMF overhead bottleneck
4	locality bottleneck	$\Longrightarrow$	locality bottleneck(R)
5	locality bottleneck(R)	$\Longrightarrow$	locality bottleneck(R,P)
6	locality bottleneck(R,P)	$\Longrightarrow$	locality bottleneck(R,P,I)
7	locality bottleneck(R,P,I)	$\Longrightarrow$	locality bottleneck(R,P,I,V)
8	locality bottleneck(R,P,I,V)	$\Longrightarrow$	thrashing(R,P,I,V) false_sharing(R,P,I,V) true_sharing(R,I,P,V,MODE) ...
9	true_sharing(R,P,I,V,MODE)	$\Longrightarrow$	true_sharing(R,P,I,V,MODE) & sequential(R1)

Table 1: Refinement rules

5.2 Refinement Rules

Refinement rules consist of a proven hypothesis and more precise hypotheses. These hypotheses can include variables that are bound to the appropriate value of the same variable in the performance bottleneck and variables that are bound to a value when the hypothesis is proven. Therefore, multiple bottlenecks can be proven from a single hypothesis with variables. The following rules are examples for refinement rules as shown in Table 1:

Rule 1:
This is the starting rule for the analysis process. It creates the basic hypothesis.

Rule 2:
The hypothesis of the existing bottleneck is refined to an unmeasured and measured overhead.

Rule 3:
It refines the hypothesis according to the different traceable overhead types.

Rule 4:
If a locality bottleneck can be proven for the whole program, this hypothesis will be refined according to the program regions. The goal is to identify program regions that have a locality bottleneck. The variable R is bound to a value when a proof rule identifies a specific region with a locality bottleneck.

Rule 5:
It determines the refinement according to the processes in the parallel program.

Rule 6:
It refines the hypothesis with respect to the instances of the region in the process.

Rule 7:
It refines the hypothesis for a specific instance. It inserts a new hypothesis for a bottleneck resulting from accesses to a single variable.

Rule 8:
It determines the possible reasons for the locality bottleneck. It shows three reasons: thrashing and coherence misses due to true and false sharing. The variable mode can be true, anti, or output, according to the access sequence write-read, read-write, and write-write responsible for the page miss.

Rule 9:
One reason for true sharing is that region R1 has not been parallelized and thus the master process accesses the variable.

5.3 Proof Rules

Proof rules contain the declaration of required performance information necessary to proof a hypothesis and appropriate predicates that represent the kernel of the rule. In addition, a confidence rate is given for each rule which is determined by the expert formulating the rule.

The first rule in Table 2 proofs if a bottleneck exists in the analyzed program.

If a bottleneck exists, the tool proofs if the unmeasured or the measured overhead dominates the total overhead. If the unmeasured part dominates, the automatic bottleneck search will stop. In the other case, described in rule three, the tool concentrates on the region with the highest total overhead.

The fourth rule determines that a bottleneck occurs in r if the region r belongs to the group with the highest total overhead time.

The next rule defines that in region r is a locality bottleneck if the locality overhead dominates in this region.

The sixth rule determines that a locality bottleneck occurs in one specific process if the process belongs to the group with the highest pagefault time for this region. The classification is calculated from the trace data by the analysis tool.

The next rule requires much more detailed information. The page fault sums have to be determined for each instance of the region in this process. The last rule proofs a locality bottleneck for a specific variable.

Table 3 outlines the proof rules for a specific reason of a locality bottleneck, namely thrashing. A vague hint for page thrashing results from the inspection of the pagefault sums of this region in all processes. If there are significant differences and some process p′ has the same high value as process p thrashing might have occurred.

A stronger hint for page thrashing is the page travel information. If two processes exchanged between each other nearly the same number of pages of that variable and there are processes with much less pagefaults, thrashing could be the reason.

Information	predicates	hypothesis	rate
execution time for the sequential and parallel program run for the whole execution in each process	Total overhead time $>$ 10% of the parallel execution time	bottleneck	100%
total overhead and all unmeasured overhead for the parallel program run, whole execution, each process.	mean of the sum of each process of the unmeasured overhead time $>$ 50% of the total overhead time.	bottleneck caused by unmeasured overhead	100%
total overhead and all measured overhead for the parallel program run, whole execution, each process.	mean of the sum of each process of the measured overhead time $>$ 50% of the total overhead time.	bottleneck caused by measured overhead	100%
statistical data for all overhead types for the parallel program run, whole execution, all regions, each process.	region r has the greatest portion of the total overhead.	bottleneck (r)	100%
instrumentation of all traceable overhead types in region r for the parallel program run, whole execution, each process.	locality overhead has the greatest portion of the total overhead.	locality bottleneck (r)	100%
read/write-page-fault-sum of r for the whole execution, each process + classification of processes according to pagefault time	Process p belong the class with the highest pagefault time	locality bottleneck (r,p)	100%
read/write-page-fault-sum of all instances of r in p + classification of instances according to pagefault time	Instance i belongs to the class with the highest pagefault time	locality bottleneck (r,p,i)	100%
read/write-page-fault-sum of r, p, i for all variables accessed in r.	v is the variable with the highest pagefault time	locality bottleneck (r,p,i,v)	100%

Table 2: Proof rules for refined hypotheses

There are two ways to proof thrashing. The first is based on the page fault sums and thus does not require much trace information. If the number of pagefaults in a process is higher than the number of pages for that variable thrashing must be the reason.

The other proof rule is based on the information of individual page faults. Tracing individual pagefaults generates more trace data than just tracing the pagefault sums but this information determines precisely which process received the pages and serviced the page faults. Thus, it is easy to proof page thrashing.

The last rule is an example of a rule that combines run-time information and analysis information. A very common situation is that an assignment statement is executed in a parallel loop and the work distribution results in assignments to the same page in different processes. If the number of pages computed in each process is additionally very small, thrashing might occur. If a large number of pages is computed in each

Information	predicates	hypothesis	rate
read/write-page-fault-sum of r and v for each process	∃ p′ in this region with a similar number of pagefaults and other processes with much less pagefaults	thrashing(r,p,v)	20%
page-travel of r and v for each process	∃ p′ which received the same number of pages from p as p from p′ and there are other processes with much less page-faults	thrashing(r,p,v)	50%
read/write-page-fault-sum of r and v for each process	the number of pagefaults is greater than the number of pages	thrashing(r,p,v)	100%
pagefault event records of r and v for p and p′	a page was exchanged multiple times among the same processes	thrashing(r,p,i,v)	100%
template distribution and variable mapping information	the region includes an assignment v(...)= ... + the work distribution and the page layout are not aligned on page boundaries + the number of pages written in a process is small	thrashing(r,p,i,v)	90%

Table 3: Proof rules for page thrashing

process it is very likely that accesses to the same page in the first iteration of a process and the last iteration of another process will not lead to page thrashing.

The rules in this section demonstrate the integration of the automatic analysis module into the current system supporting manual performance analysis. The existing environment provides all the information required to apply the rules and to detect performance bottlenecks.

6 Summary

This article outlined the performance analysis environment for SVM-Fortran. OPAL supports the whole incremental analysis procedure via an interactive interface, but still the user has to decide which information to trace in the next program run to be able to identify performance bottlenecks. For the most common performance bottlenecks this analysis can be automated. The article presented a design to implement this automation.

Currently, several applications have been parallelized with SVM-Fortran. These experiments lead to a deep understanding of the performance bottlenecks of SVM systems and enabled us to start formulating the rules for the automatic performance analysis module.

References

[1] K. Li, *IVY: A Shared Virtual Memory System for Parallel Computing*, International Conference on Parallel Processing (ICPP'88), Vol II, pp. 94-101, 1988

[2] M.T. Heath, J.A. Etheridge. *Visualizing the performance of parallel programs*, IEEE Software, 8(5), pp.29-29, 1991

[3] D.A. Reed, R.A. Aydt, T.M. Madhyastha, R.J. Noe, K.A. Shields, B.W. Schwartz, *An Overview of the Pablo Performance Analysis Environment*, Technical Report, University of Illinois, Department of Computer Science, 1992

[4] W.E. Nagel, A. Arnold, M. Weber, H-C. Hoppe, K. Solchenbach, *VAMPIR: Visualization and Analysis of MPI Resources*, Supercomputer 63, Vol. 12, No. 1, pp. 69-80, 1996

[5] Cray Research, *Introducing the MPP Apprentice Tool*, Cray Manual IN-2511, 1994

[6] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall, *The Paradyn Parallel Performance Measurement Tools*, IEEE Computer, Vol. 28, No. 11, pp. 37-46, 1995

[7] R. Berrendorf, M. Gerndt, *SVM-Fortran Reference Manual Version 1.4*, Internal Report KFA-ZAM-IB-9510, Central Institute for Applied Mathematics, Research Centre Jülich, 1995

[8] M. Gerndt, R. Berrendorf, *Parallelizing Applications with SVM-Fortran Proceedings of the HPCN'95*, Milan, LNCS 919, pp. 793 - 798, 1995

[9] HPFF, *High Performance Fortran Language Specification, Version 1.1, November 1994* Rice University, Houston, Texas 1994

[10] F. Bodin, L. Kervella, T. Priol, *Fortran-S: A Fortran Interface for Shared Virtual Memory Architectures*, International Conference on Supercomputing 1993, Portland, pp. 274-283, 1993

[11] Kendall Square Reserach Corporation, *KSR Fortran Programming Manual*, Kendall Square Research, 1992

[12] S. Özmen, *SAM: Performance-Analyse-Monitor für SVM-Fortran*, Diploma Thesis, RWTH Aachen, 1995

[13] M. Gerndt, *Performance Analysis Environment for SVM-Fortran Programs*, Internal Report KFA-ZAM-IB-9417, Central Institute for Applied Mathematics, Research Centre Jülich, 1994

[14] M. Gerndt, A. Krumme, S. Özmen, *Performance Analysis for SVM-Fortran with OPAL*, Proceedings Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'95), Athens, Georgia, pp. 561-570, 1995